A Framework for the Detection and Resolution of Aspect Interactions

Rémi Douence[1,⋆], Pascal Fradet[2], and Mario Südholt[1,*]

[1] École des Mines de Nantes/Inria, Nantes, France
www.emn.fr/{douence,sudholt}
[2] Irisa/Inria, Rennes, France
www.irisa.fr/lande/fradet

Abstract. Aspect-Oriented Programming (AOP) promises separation of concerns at the implementation level. However, aspects are not always orthogonal and aspect interaction is an important problem. Currently there is almost no support for the detection and resolution of such interactions. The programmer is responsible for identifying interactions between conflicting aspects and implementing conflict resolution code.
In this paper, we propose a solution to this problem based on a generic framework for AOP. The contributions are threefold: we present a formal and expressive crosscut language, two static conflict analyses and some linguistic support for conflict resolution.

1 Introduction

Separation of concerns is a valuable structuring principle for the development of software systems. Aspect-Oriented Programming (AOP) [8] promises a systematic treatment of concern separation at the implementation level. Once concerns are expressed separately in terms of different aspect definitions, one of the most fundamental problems of AOP is that of interaction between aspects, i.e., conflicts between aspects which are not orthogonal [4]. There is almost no support for the treatment of aspect interactions: the programmer is responsible for identifying interactions between conflicting aspects and for implementing conflict resolution code.

We believe that the treatment of aspect interactions should be separated from the definition of the aspects themselves. We therefore propose a three-phase model for multi-aspect programming:

1. Programming. The aspects which are part of an application are written independently, possibly by different programmers.
2. Conflict analysis. An automatic tool detects interactions among aspects and returns informative results to the programmer.

⋆ Partially funded by the EU project "EasyComp" (www.easycomp.org), no. IST-1999-014191.

D. Batory, C. Consel, and W. Taha (Eds.): GPCE 2002, LNCS 2487, pp. 173–188, 2002.
© Springer-Verlag Berlin Heidelberg 2002

3. Conflict resolution. The programmer resolves the interactions using a dedicated composition language. The result of this phase can be checked once again as in phase 2.

The main objective of this paper is to provide support for this three-phase process. Our solution is based on a generic framework for AOP, which is characterized by a very expressive crosscut language, static conflict analyses and linguistic support for conflict resolution.

In Section 2, we formally define a general model for AOP which (conceptually) relies on a monitor observing the execution trace. Aspects consist of crosscuts matching regular expressions (i.e., sequences) of execution points, which define where the execution of the base program is modified, and inserts, which define the aspect code to be executed. Aspect weaving is modeled as an execution monitor recognizing crosscuts and applying inserts. The interactions treated in this paper occur when two crosscuts match the same point in the execution trace. In Section 3, we propose two different analyses detecting aspect interactions, independently of the base program or *w.r.t.* a specific base program. Section 4 proposes some linguistic support for the resolution of conflicts caused by aspect interactions. In particular, we introduce commands making explicit the composition of several inserts at the same execution point. We also present commands to control the visibility of aspects *w.r.t.* other aspects. These commands are taken into account by aspect transformation so that interaction analyses can still be applied to check that conflicts have been effectively resolved. We conclude by a brief review of related work and future research directions.

This paper provides a generic model for AOP that does not rely on any specific programming language. In order to provide more intuition, we illustrate our different concepts by instantiating the framework to AspectJ [9]. We assume a basic familiarity with AOP [8] in general and AspectJ in particular.

2 Framework

We model weaving as a dynamic monitor, observing the execution of the program and inserting instructions according to execution states. An aspect specifies which instructions to insert at which execution state. This study is made within a generic formal framework. In particular, we do not rely upon a particular programming language and consider a very expressive crosscut language.

We first present our model of program execution. We then introduce our aspect language that is based on crosscuts, inserts and composition operators. The operators support expressive aspect definitions and enable modeling of aspect interactions. Finally, we describe the weaver, that is to say, how executions are monitored, i.e., observed and woven.

2.1 Observable Execution and Join Points

The relevant part of an execution for weaving is called the observable execution trace. We define it using a transition relation ($\rightarrow$) between observable execution states. A $\rightarrow$-step may represent a sequence of actual execution steps. The

relation $\rightarrow$ can be defined on the basis of a small-step semantics [11] of the base programming language. Observable states are *configurations* of the form (j, P, σ), where j, the current join point, is an abstraction of the (static) program P and the (dynamic) execution state σ. Join points are terms which can be matched against crosscuts, i.e., term patterns. Their nature can be syntactic (e.g., instructions) but also semantic (e.g., dynamic values).

The entry and exit of a program are denoted by two special join points: $\downarrow$ and $\uparrow$, respectively. The observable execution trace of a program with an initial state σ_0 is then of the form:

$$(\downarrow, P, \sigma_0) \rightarrow \ldots \rightarrow (j_i, P, \sigma_i) \rightarrow \ldots$$

If the reduction terminates, there exists a σ_n such that $(\downarrow, P, \sigma_0) \stackrel{*}{\rightarrow} (\uparrow, P, \sigma_n)$, where $\stackrel{*}{\rightarrow}$ denotes the transitive, reflexive closure of $\rightarrow$.

AspectJ: In AspectJ, join points denote, among others, method calls, field accesses and exception handler executions. They are represented at run time by a variable `thisJoinPoint` that contains static information (e.g., method signatures, source locations) as well as dynamic information (e.g., values of the receiver and arguments of a call). For example, when a point object (whose address is 4711) is moved to the origin through a call occurring in a line object (at address 1213), the corresponding join point can be modeled in our framework by a term:

```
call(void Point.move(int,int), within(Line),
     this(Line,1213), target(Point,4711), args(0,0))
```
■

2.2 The Aspect Language

The basic constituents of aspects are rules of the form

$$C \triangleright I$$

where C is a *crosscut* and I an *insert*. The insert I is a program that is executed whenever the crosscut C matches the current join point. Rules are combined into aspects using three operators (sequence, repetition and choice).

Crosscuts. To achieve highest expressiveness, crosscuts would be defined as arbitrary functions matching join points. However, this is too general for our purposes: we consider a more specific yet expressive crosscut language in which checking interactions is feasible.

Let us define terms as finite trees of the form

$$T ::= \mathtt{f}\ T_1 \ldots T_n \mid x$$

where `f` is an n-ary ($n \geq 0$) symbol and x is a variable. A term can be seen as a pattern to be matched on join points. The symbol `f` can represent a syntactic element of the programming language or, more generally, an information contained within join points. Note that our aspect language is generic and can be used to define more specialized term languages (e.g., one for AspectJ).

A crosscut is made of conjunctions, disjunctions and negations of terms:

$$C ::= T \mid C_1 \wedge C_2 \mid C_1 \vee C_2 \mid \neg C$$

For example, using a more concrete syntax than the abstract trees denoting terms, the crosscut matching calls to a function `g` where one of the two arguments is a constant `a` can be written $\mathtt{g}(x, \mathtt{a}) \ \vee \ \mathtt{g}(\mathtt{a}, x)$.

The formulas used to express crosscuts belong to the so-called quantifier free equational formulas [3]. Whether such a formula has a solution is decidable. This is one of the key properties making the analyses described in Section 3 feasible.

The application of a crosscut to a join point j is written $C\ j$. It amounts to solving the formula obtained by replacing each term T in C by the equation $j = T$. If a crosscut does not match the program point (i.e., the formula has no solution) then we write $C\ j = \mathtt{fail}$. If the crosscut matches the program point then we write $C\ j = \phi$ where ϕ is a substitution mapping the variables of the crosscut to their unique solution (variables with several solutions do not appear in ϕ)

We use *false* for the crosscut which does not match any join point and *true* for the crosscut that matches all join points. Let z be a fresh variable then *false* can be defined by the crosscut $z \wedge \neg z$ and *true* by z.

AspectJ: AspectJ's crosscuts ("pointcuts") are very close to the crosscut language we introduced. They are defined as terms containing variables ranging over values of programs and wildcards. They may be combined using the same logical operators (`&&`, `||`, `!`). For example, moving points could be tracked in AspectJ with the crosscut definition:

```
pointcut moving(Point p, int x):target(p) && call(void move(x,*))
```

This crosscut can be translated in our framework into the pattern:

$$\mathtt{call}(\mathtt{void\ Point.move}(\mathtt{int}, w_0), \mathtt{within}(w_1), \mathtt{this}(w_2), \mathtt{target}(\mathtt{Point}\ p), \mathtt{args}(x, w_3))$$

where fresh variables w_i express wildcards or irrelevant information. ■

Inserts. An insert I is a term as defined above. The intuition behind a rule $C \triangleright I$ is that when the crosscut matches the current join point, i.e., $C\ j = \phi$, then ϕI is executed. Hence, $C\ j$ must yield a substitution binding all the variables of I. Any specific aspect language must ensure that ϕI is a always a valid piece of code (in particular, that it does not contain undefined term variables). In the remainder of this paper we assume that all ϕI are valid.

At some places, we use the special insert `skip` that represents an instruction doing nothing. We write *always* for the rule $\mathit{true} \triangleright \mathtt{skip}$ that matches any join point and does nothing and *never* for the rule $\mathit{false} \triangleright \mathtt{skip}$ that does not match any join point.

AspectJ: AspectJ's inserts ("advice") are defined as Java code to be executed when a crosscut matches. As in our language, they may refer to the values bound to the variables occurring in the corresponding crosscut. ■

Aspects. In order to define aspects, we use a syntax similar to process calculi such as CSP. An aspect is defined by the following grammar:

$$\begin{array}{lll} A ::= & \mu a.A & \textit{; recursive definition} \\ \mid & C \rhd I \; ; \; A & \textit{; sequence} \\ \mid & C \rhd I \; ; \; a & \textit{; end of sequence} \\ \mid & A_1 \; \Box \; A_2 & \textit{; choice} \end{array}$$

An aspect is either

- The recursive definition of an aspect $\mu a.A$ which is equivalent to the aspect A where all the occurrences of the variable a are replaced by $\mu a.A$.
- A sequence $C \rhd I \; ; \; X$, where X is an aspect or a variable. These linear sequences always end with a variable. This is needed to ensure that aspects are regular (finite state). If the crosscut C matches the current join point, then X becomes the aspect to be woven. We consider that as soon as a rule has matched a join point, it terminates. An aspect trying to apply $C \rhd I$ throughout the execution can be expressed as

  $$\mu a.C \rhd I \; ; \; a$$

  This aspect does not evolve during the execution: such an aspect is called *stateless*. An aspect applying $C \rhd I$ only once can be expressed as

  $$C \rhd I \; ; \; (\mu a.\textit{never} \; ; \; a)$$

  Indeed, as soon as $C \rhd I$ is applied, the weaver will try to apply *never*. This is an instance of an aspect evolving according to the join points encountered. Their implementation must use some kind of state to represent this evolution. We use the term *stateful* to refer to this general form of aspects.
- A choice construction $A_1 \; \Box \; A_2$ which chooses the first aspect that matches a join point (the other is thrown away). If both match the same join point, A_1 is chosen. For example, the aspect trying to apply $C \rhd I$ only on the current join point and doing nothing afterward can be expressed as

  $$(C \rhd I \; ; \; (\mu a.\textit{always} \; ; \; a)) \; \Box \; (\mu a.\textit{always} \; ; \; a)$$

  If C matches the current join point, the weaver chooses the first aspect, executes the insert I and the aspect becomes $\mu a.\textit{always} \; ; \; a$ that keeps doing nothing. Otherwise, the weaver chooses the second recursive aspect which is $\mu a.\textit{always} \; ; \; a$ as well.

Recursive definitions, sequencing, and choices allow the specification of finite state aspects which evolve according to the join points encountered. For example, a security aspect that logs file accesses (calls to `read`) during a session (from a call to `login()` until a call to `logout()`) can be expressed as

$$\mu a_1.\texttt{login}() \rhd \texttt{skip} \; ; \; \mu a_2.(\texttt{logout}() \rhd \texttt{skip} \; ; \; a_1) \; \Box \; (\texttt{read}(x) \rhd \texttt{addLog}(x) \; ; \; a_2)$$

where x denotes the name of the accessed file.

Aspect composition. Aspects addressing different issues (such as debugging and profiling) are composed using a parallel operator $\|$. Typically, the weaver takes a parallel composition of n aspects $A_1 \| \ldots \| A_n$ and tries to apply each of them at each join point. The parallel operator is non-deterministic. For example, the composition

$$(\mu a.C_1 \rhd I_1 \; ; \; a) \; \| \; (\mu a.C_2 \rhd I_2 \; ; \; a)$$

inserts I_1 (resp. I_2) if C_1 (resp. C_2) matches the current join point. When C_1 and C_2 match the same join point, it is not specified whether I_1 is executed before I_2 or vice versa. In this case, we say that $(\mu a.C_1 \rhd I_1 \; ; \; a)$ and $(\mu a.C_2 \rhd I_2 \; ; \; a)$ interact.

AspectJ: AspectJ's aspects are rules (in the sense above) and they are repeatedly applied throughout the program execution. They can be expressed in our framework as $\mu a.C \rhd I \; ; \; a$. Several aspects are composed in parallel ($\|$). Therefore, they may match the same join point and interact. In AspectJ, conflicts are resolved based on user annotations ("aspect domination") and the hierarchy of aspects. However, when two aspect are unrelated *w.r.t.* the domination or hierarchy relations, the ordering of inserts is undefined.

Composition of aspects by means of sequence and choice operators have no equivalent in AspectJ. The user must manually instrument advices with a state and appropriate conditions in order to simulate them. So, our crosscut language is more expressive than the crosscut language of AspectJ. ■

2.3 Weaving

In order to describe aspect weaving we need to introduce several auxiliary functions.

The **sel** function takes an aspect and extracts the rule to apply at the current join point j.

$$\begin{array}{lcll}
\textbf{sel}\ j\ (\mu a.A) & = & \textbf{sel}\ j\ A & \\
\textbf{sel}\ j\ (C \rhd I \; ; \; A) & = & \emptyset & \text{if } C\ j = \texttt{fail} \\
 & = & \{C \rhd I\} & \text{otherwise} \\
\textbf{sel}\ j\ (A_1 \ \Box \ A_2) & = & \textbf{sel}\ j\ A_1 & \text{if } \textbf{sel}\ j\ A_1 \neq \emptyset \\
 & = & \textbf{sel}\ j\ A_2 & \text{otherwise}
\end{array}$$

The following rule extends **sel** to the parallel composition of several aspects

$$\textbf{sel}\ j\ (A_1 \| \ldots \| A_n) = (\textbf{sel}\ j\ A_1) \cup \ldots \cup (\textbf{sel}\ j\ A_n)$$

The **next** function represents the evolution of an aspect after the current join point j. It takes a composite aspect and yields the aspect to be applied to the next join point.

$$\begin{array}{lcll}
\textbf{next}\ j\ (\mu a.A) & = & \textbf{next}\ j\ A[\mu a.A/a] & \\
\textbf{next}\ j\ (C \rhd I \; ; \; A) & = & C \rhd I \; ; \; A & \text{if } C\ j = \texttt{fail} \\
 & = & A & \text{otherwise} \\
\textbf{next}\ j\ (A_1 \ \Box \ A_2) & = & \textbf{next}\ j\ A_1 & \text{if } \textbf{sel}\ j\ A_1 \neq \emptyset \\
 & = & \textbf{next}\ j\ A_2 & \text{if } \textbf{sel}\ j\ A_2 \neq \emptyset \\
 & = & (A_1 \ \Box \ A_2) & \text{otherwise}
\end{array}$$

It is extended to the parallel composition of several aspects using the rule

$$\textbf{next}\ j\ (A_1 \| \ldots \| A_n) = (\textbf{next}\ j\ A_1) \| \ldots \| (\textbf{next}\ j\ A_n)$$

The woven execution is performed relative to a composite aspect A (see Figure 1). The transition relation $\Longrightarrow$ represents the woven execution. It is defined by the application of the monitor followed by a standard execution step and yields the aspect (**next** j A) to be applied to the following join point. At each join point, the applicable rules are selected (**sel** j A). The monitor (relation $\models\!\!\Longrightarrow$) applies the selected rules in no specific order: if the crosscut of the current rule matches the current join point, the corresponding substitution is applied to the insert and ϕI is executed.

Woven execution

$$\frac{[j, P, \sigma]^{\textbf{sel}\ j\ A} \overset{*}{\models\!\!\Longrightarrow} \sigma_a \qquad (j, P, \sigma_a) \rightarrow (j', P, \sigma')}{(A, j, P, \sigma) \Longrightarrow (\textbf{next}\ j\ A, j', P, \sigma')}$$

Monitor

$$[j, P, \sigma]^{\emptyset} \models\!\!\Longrightarrow \sigma \quad [\text{end}]$$

$$\frac{\mathcal{S} = \{C \rhd I\} \cup \mathcal{S}' \qquad C\ j = \phi \qquad (\downarrow, \phi I, \sigma) \overset{*}{\rightarrow} (\uparrow, \phi I, \sigma')}{[j, P, \sigma]^{\mathcal{S}} \models\!\!\Longrightarrow [j, P, \sigma']^{\mathcal{S}'}} \ [\text{match}]$$

Fig. 1. Dynamic weaving of silent inserts

Note that we use $\overset{*}{\rightarrow}$ to reduce inserts. This implies that inserts are not subject to weaving. In this case, we say that inserts are *silent*. Figure 2 formalizes another option which uses $\overset{*}{\Longrightarrow}$ to execute inserts. This makes inserts *visible* to the weaver. Since the composition of aspects may evolve during the execution of visible inserts, it must be passed to and returned by the monitor.

The programmer may want to choose whether the (inserts of an) aspect A_1 is visible for the weaving of aspect A_2. We come back to this issue in Section 4.

Note that since no specific order of application of aspects has been specified, weaving may be non-deterministic. This situation arises when aspects interact, that is to say when **sel** j A returns a set of at least two rules. Detecting and resolving such cases is the objective of the next two sections.

AspectJ: In AspectJ, aspects are silent *w.r.t.* one another and the base program is visible to all aspects. So, the first version of the weaver should be used. However, when an advice calls a method of the base program, the woven version of the method is executed in AspectJ. Indeed, static weaving (based on program

Woven execution

$$\frac{[j,P,\sigma]^{\mathbf{sel}\ j\ A}_{\mathbf{next}\ j\ A} \overset{*}{\Longmapsto} (\sigma_a, A') \quad (j,P,\sigma_a) \rightarrow (j',P,\sigma')}{(A,j,P,\sigma) \Longrightarrow (A',j',P,\sigma')}$$

Monitor

$$[j,P,\sigma]^{\emptyset}_{A} \Longmapsto (\sigma, A) \quad [\text{end}]$$

$$\frac{\mathcal{S} = \{C \rhd I\} \cup \mathcal{S}' \quad C\ j = \phi \quad (A,\downarrow,\phi I,\sigma) \overset{*}{\Longrightarrow} (A',\uparrow,\phi I,\sigma_a)}{[j,P,\sigma]^{\mathcal{S}}_{A} \Longmapsto [j,P,\sigma_a]^{\mathcal{S}'}_{A'}} \ [\text{match}]$$

Fig. 2. Dynamic weaving of visible inserts

transformation rather than execution monitoring) makes this natural. This behavior does not correspond exactly to either of the two weaver definitions given above.

In the beginning of this paper, we introduced method-call join points. On occurrence of such a join point, our weaver definitions first execute the insert followed by the base execution after the join point. This behavior corresponds to AspectJ's `before`-advices. In order to take into account AspectJ's advice qualifier `after`, a new kind of join point must be introduced which represents when a method returns (the insert is executed *after* the method returns). In our framework, the weaver cannot skip portions of the base program execution. So, we can only model (by means of `before` and `after`) `around` advices that call `proceed` as part of the advice. ■

3 Aspect Interactions

One of our goals is to detect when the naive parallel composition of aspects does not guarantee a deterministic weaving. We say that two aspects are independent if they do not interact (i.e., none of their crosscuts may match the same join point). Independence of two aspects is a sufficient condition to ensure that weaving is well-defined: in this case, they can be woven in any order. On the opposite, dependent aspects require the programmer to resolve the interactions.

We distinguish between two notions of independence:

- *Strong independence* does not depend on the program to be woven. The aspects are independent for all programs. The advantage of this property is that it does not have to be checked after each program modification.
- *Independence w.r.t. a program* takes into account the possible sequences of join points generated by the program to be woven. The advantage of this property compared to strong independence is that it is a weaker condition to enforce.

Note that independence (strong or *w.r.t.* a program) is a sufficient but not a necessary condition. If two crosscuts C and C' match the same join point but their corresponding inserts I and I' commute (i.e., executing I then I' is equivalent to executing I' then I) then the woven execution remains deterministic.

3.1 Strong Independence

We start by defining strong independence for crosscuts.

Definition 1. *Two crosscuts C and C' are said to be* strongly independent *if $C \wedge C'$ has no solution.*

This ensures that the two crosscuts can never match the same join point. When crosscuts are simple patterns (i.e., terms) strong independence amounts to checking that they are not unifiable. When the crosscuts involve negations, conjunctions and disjunctions, $C \wedge C'$ is an equational formula and remains solvable [3].

The algorithm to check strong independence of aspects is based on the laws shown in Figure 3. The algorithm, which is similar to the algorithm for finite-state product automata, terminates due to the finite-state nature of our aspects (the *(un)fold* law is used to fold already encountered aspects). We only describe here its essential properties.

[*(un)fold*]	$\mu a.A = A[\mu a.A/a]$
[*assoc*]	$(A_1 \,\Box\, A_2) \,\Box\, A_3 = A_1 \,\Box\, (A_2 \,\Box\, A_3)$
[*commut*]	$(C_1 \rhd I_1 \,;\, A_1) \,\Box\, (C_2 \rhd I_2 \,;\, A_2) = (C_2 \rhd I_2 \,;\, A_2) \,\Box\, (C_1 \rhd I_1 \,;\, A_1)$ if $C_1 \wedge C_2$ has no solution
[*elim*$_1$]	$C \rhd I = \mathit{false} \rhd I$ if C has no solution
[*elim*$_2$]	$(\mathit{false} \rhd I \,;\, A_1) \,\Box\, A_2 = A_2$
[*elim*$_3$]	$\mathit{false} \rhd I \,;\, C_1 \rhd I_1 \,;\, A = \mathit{false} \rhd I \,;\, A$
[*priority*]	$(C_1 \rhd I_1 \,;\, A_1) \,\Box\, (C_2 \rhd I_2 \,;\, A_2) = (C_1 \rhd I_1 \,;\, A_1) \,\Box\, (C_2 \wedge \neg C_1 \rhd I_2 \,;\, A_2)$
[*propag*]	let $A = (C_1 \rhd I_1 \,;\, A_1) \,\Box\, \ldots \,\Box\, (C_n \rhd I_n \,;\, A_n)$ and $A' = (C'_1 \rhd I'_1 \,;\, A'_1) \,\Box\, \ldots \,\Box\, (C'_m \rhd I'_m \,;\, A'_m)$ then $A \parallel A' = \Box_{i=1..n}^{j=1..m} C_i \wedge C'_i \rhd (I_i \bowtie I'_j) \,;\, (A_i \parallel A'_j)$ $\Box_{i=1..n} C_i \rhd I_i \,;\, (A_i \parallel A')$ $\Box_{j=1..m} C'_j \rhd I'_j \,;\, (A \parallel A'_j)$

Fig. 3. Laws for aspects

The main law is *propag* which propagates the parallel operator inside the aspect definition. It produces a sequence of choices made of all the possible pairs

of crosscuts from A and A' and all the single crosscuts of A and A' independently. Conflicts are represented using the non-deterministic function $(I_1 \bowtie I_2)$ which returns either $I_1;I_2$ or $I_2;I_1$ (where ";" denotes the sequencing operator of the programming language). The law $elim_1$ uses the algorithm of [3] to check if a crosscut has no solution in which case the crosscut is replaced by *false*. The laws $elim_2$ and $elim_3$ remove unreachable parts of an aspect. The *priority* accounts for the priority rules implicit in the choice operator. This makes the analysis more precise (e.g., using this law and $elim_2$, $(true \rhd I_1 \; ; \; A_1) \; \Box \; A_2$ can be rewritten into $(true \rhd I_1 \; ; \; A_1))$. The laws *assoc*, *commut* and *(un)fold* serve to rewrite an aspect so that the other laws can be applied.

Definition 2. *Two aspects A and A' are said to be* strongly independent *if $A \parallel A'$ can be expressed as a single aspect (i.e. without $\bowtie$ and $\parallel$).*

For example, the parallel composition

$$A \parallel A' = (\mu a.C \rhd I \; ; \; a) \parallel (\mu a.C' \rhd I' \; ; \; a)$$

can be rewritten using *(un)fold* twice, *propag* and *fold* again into

$$\mu a.(C \wedge C' \rhd (I \bowtie I') \; ; \; a) \; \Box \; (C \rhd I \; ; \; a) \; \Box \; (C' \rhd I' \; ; \; a)$$

If C and C' are independent then, using $elim_1$ and $elim_2$, it can be rewritten into

$$\mu a.(C \rhd I \; ; \; a) \; \Box \; (C' \rhd I' \; ; \; a)$$

a deterministic, sequential aspect.

AspectJ: As explained in the previous section, each rule in AspectJ is of the form: $\mu a.C_i \rhd I_i \; ; \; a$. So, the analysis of strong independence of two aspects boils down to check the independence of two crosscuts C_1 and C_2. For example, the analysis detects that the two following crosscuts are unifiable and therefore not strongly independent:

```
call(void *.move(*, int)) and call(* Point.*(int, *))
```

■

3.2 Independence *w.r.t.* a Program

Strong independence may be too strong a condition. It is sufficient to check independence *w.r.t.* the set of possible observable execution traces of a program. These traces depend on whether inserts are visible or not. We first consider the case of silent aspects, i.e., inserts which are not subject to weaving.

The precise set of execution traces is not statically computable. We assume that we have a finite approximation taking the form of a finite set of join points $\mathcal{J}(P)$ and a function

$$\mathbf{step}_P : \mathcal{J}(P) \rightarrow \mathcal{P}(\mathcal{J}(P))$$

giving for each join point a superset of the possible successors. When the join points are purely syntactic (and when new syntax cannot be dynamically created as it is possible, e.g., using Lisp's backquote-construction), then a possible approximation is to take all the join points of the program for $\mathcal{J}(P)$ and

$\mathbf{step}_P\ j = \mathcal{J}(P)$ for every join point. This crude approximation (all join points can follow each join point) is sufficient for stateless aspects. For stateful aspects, we may rely on techniques based on control flow to get more precise approximations. To be safe, such an analysis must take into account the impact that inserts may have on the control flow of the base program.

We can specialize the parallel composition of aspects *w.r.t.* the possible sequences of join points. The function I_w formalizes such a specialization. In the following definition, we assume that A is a parallel composition of two aspects (i.e., at most two crosscuts can match a join point).

$$
\begin{array}{lll}
Wrti(A,j) & = & \\
\textit{if} & \mathbf{sel}\ j\ A = \emptyset & \textit{then}\ \ \Box_{j' \in (\mathbf{step}_P\ j)}\ I_w(A,j') \\
\textit{else if} & \mathbf{sel}\ j\ A = \{C \rhd I\} & \textit{then}\ \ C \rhd I\ ;\ \Box_{j' \in (\mathbf{step}_P\ j)} I_w(\mathbf{next}\ j\ A, j') \\
\textit{else if} & \mathbf{sel}\ j\ A = \{C \rhd I, C' \rhd I'\} & \\
 & \textit{then}\ C \wedge C' \rhd (I \bowtie I')\ ;\ \Box_{j' \in (\mathbf{step}_P\ j)} I_w(\mathbf{next}\ j\ A, j') &
\end{array}
$$

The process starts with an aspect and the entry of the program $\downarrow$. The crosscuts matching the current join point are extracted (**sel**). The process is iterated with the new aspects (computed by **next**) and all possible successors (given by **step**). The resulting aspects are combined with the choice operator. Due to the finite-state nature of aspects and join points, there are only a finite number of reachable pairs (A, j) and I_w terminates. The laws *(un)fold*, $elim_1$ and $elim_2$ are then used to simplify the expression.

Definition 3. *Two aspects A and A' are independent w.r.t. a program P if $I_w(A \parallel A', \downarrow)$ can be expressed as a single aspect (i.e. without $\bowtie$ and $\parallel$).*

If inserts are visible, join points generated by inserts must be taken into account by the control flow analysis. This requires to compute an approximation of the set of join points and the possible insertions.

Note that since (visible) inserts produce new syntax dynamically, it is even possible that the weaving process loops and introduces an unbounded number of new join points. For instance, the following profiling aspect repeatedly crosscuts any method call in order to increment a counter:

$$\mu a.(\texttt{call}(x.y(z)) \rhd \texttt{Profiler.incrCall}()) \ ;\ a$$

This aspect crosscuts its own insert and weaving loops: the first method call of the base program is crosscut, so `Profiler.incrCall()` is called, which is itself crosscut, etc.

AspectJ: The two crosscuts

`call(void *.move(*, int))` and `call(* Point.*(int, *))`

are independent w.r.t. to programs which do not contain call sites corresponding to the unification of the two patterns (i.e., `call(void Point.move(int, int))`). ■

3.3 Semantic Crosscuts

Most of the crosscuts we have considered so far match syntactic information such as method calls, etc. As already suggested, crosscuts can also match semantic information, such as dynamic values. For instance, the rule that matches only join points where the first argument of `move` is zero can be expressed as x_1.`move`(0,x_3)$\triangleright I$. In general, the dynamic information must be encoded as terms in join points. For example, let us consider the crosscut that matches method calls to `move` if the value of the first argument of the call is even. A simple solution would be to instrument the insert with a test, such as

$$x_1.\texttt{move}(x_2, x_3) \triangleright \texttt{if (even } x_2\texttt{) then } I$$

The drawback of this approach is that the conflict analysis is not able to take the parity condition into account. A more precise solution is to encode the parity information in the join point model. For example, we may enhance the join point model of AspectJ with a constructor `Even` to denote the parity of the arguments of a call. The join point

```
call(void Point.move(int,int),...,args(2,3),Even(true,false))
```

makes explicit that the first and second argument of the call to `move` are respectively even and odd.

Both independence analyses can take dynamic information into account. The interaction analysis *w.r.t.* a program can perform a static analysis of the semantic properties to improve its precision.

AspectJ: AspectJ provides a construction `cflow(C1) && C2`. It can be expressed in our framework as:

$$\mu a_1.C_1 \triangleright \texttt{skip}\ ;\ \mu a_2.(RetC_1 \triangleright \texttt{skip}\ ;\ a_1)\ \square\ (C_2 \triangleright I\ ;\ a_2)$$

where C_1 defines a method call join point and $RetC_1$ defines the corresponding method-return join point. This definition can be read as: "between C_1 and $RetC_1$, occurrences of C_2 trigger execution of I". However, this definition is only valid when the method denoted by C_1 is not recursive. In general, such a crosscut is semantic. In AspectJ, `cflow`'s implementation requires a stack in order to count (i.e., store) pending calls to C_1.

Similarly to the parity property above, a solution is to encode in the join point the presence/absence of (at least) one pending call in the execution stack for every method in the program (e.g., using a bit vector). The conflict analysis *w.r.t.* a program could approximate this information using static analysis. For example, when `cflow(`C`)` is involved, we can safely assume that there is at least one call to C in the stack for every join point in the set of reachable methods from C.

Analysis of strong independence cannot make assumption about the call graph of the application. So, we must assume that every method has pending calls in the stack, and when `cflow(`C_1`) && `C_2 is involved, the analysis can only consider C_2. However, there are special cases of crosscuts involving `cflow` (such as `cflow(`C_1`) && `C_2 and `!cflow(`C_1`) && `C_3) which can be shown strongly independent. ■

4 Support for Conflict Resolution

When no conflicts have been detected, the parallel composition of aspects can be woven without modifications. Otherwise, the programmer must get rid of the nondeterminism by making the composition more precise. We present here some linguistic support aimed at resolving interactions. A first kind of commands serves to specify how inserts compose. A second kind allows the user to control visibility of inserts by restricting the scope of aspects. We describe a collection of useful commands which is, however, not meant to be complete.

4.1 Composition of Inserts

The conflict analyses of Sections 3.1 and 3.2 both return aspects as results. The occurrences of rules of the form $C \rhd (I_1 \bowtie I_2)$ indicate potential interactions.

These interactions can be resolved one by one. For each $C \rhd (I_1 \bowtie I_2)$, the programmer may replace each rule $C \rhd (I_1 \bowtie I_2)$ by $C \rhd I_3$ where I_3 is a new insert which combines I_1 and I_2 in some way.

This option is flexible but can be tedious. Instead of writing a new insert for each conflict, the programmer may indicate how to compose inserts at the aspect level. We propose parallel operators of the form $\|_f$ to indicate that whenever a conflict occurs in the composition $A \|_f A'$, the corresponding inserts must be composed using f. Of course, these operators can be combined to compose several aspects (e.g., $A \|_f (A' \|_g A'')$)

For example, when an insert I_1 of A_1 conflicts with an insert I_2 of A_2,

- $A_1 \|_{seq} A_2$ inserts $I_1;I_2$, (where ";" denotes the sequencing operator of the programming language).
- $A_1 \|_{fst} A_2$ inserts I_1 only.

Let us consider two aspects whose composition produces conflicts: $A_{encryption}$ crosscuts some method calls and encodes their arguments and $A_{logging}$ logs some method calls.

- $A_{logging} \|_{seq} A_{encryption}$ generates logs for super users by logging method calls with original arguments,
- $A_{encryption} \|_{seq} A_{logging}$ generates logs for users by logging method calls with possibly encrypted arguments,
- $A_{encryption} \|_{fst} A_{logging}$ generates logs for basic users where the encrypted methods do not appear.

Another class of commands concerns spurious conflicts. Indeed, when inserts commute in a conflict (e.g., one of the insert is `skip`), the inserts can be executed in any order. The programmer may use the command I_1 `commute` I_2 to allow the analyzer to produce an arbitrary sequence of I_1 and I_2.

All these assertions can be taken into account by the analyzer. If there are still conflicts, the analyzer warns the programmer that the composition is not

yet completely specified. The process can be iterated until the composition of aspects can be rewritten into a single deterministic aspect.

AspectJ: In AspectJ, conflicting advice can be ordered with `dominate` which is equivalent to $\|_{seq}$. The programmer must manually implement other compositions. ■

4.2 Scope of Aspects

In the weaver defined in Figure 2, the inserts are subject to weaving. This option is conflict-prone. In order to control visibility, we propose a notion of scope for aspects. The command

$$\texttt{scope}\ id\ Idset\ A$$

declares an aspect A with name id which can match only join points coming from an aspect whose name belongs to $Idset$. The join points of inserts are supposed to be tagged by the name of the aspects the inserts belong to. The join points of the base program are supposed to be tagged by `base`.

Scope declarations allows us to define aspects of aspects. For instance, it becomes possible to compose a profiling aspect with a security aspect in order to evaluate the cost of security tests in an application:

$$(\texttt{scope sec}\ \{\texttt{base}\}\ A_{security}) \parallel (\texttt{scope prof}\ \{\texttt{sec}\}\ A_{profiling})$$

In order to profile both the security aspect and the base application, we should use the following declaration

$$(\texttt{scope sec}\ \{\texttt{base}\}\ A_{security}) \parallel (\texttt{scope prof}\ \{\texttt{base}, \texttt{sec}\}\ A_{profiling})$$

We pointed out in Section 3.2 that visible inserts may lead to an infinite loop in the weaver. Preventing cycles in the scope declarations (e.g., an aspect cannot see its own inserts) is sufficient to ensure that such non-terminating weaving never occurs.

AspectJ: As mentioned at the end of Section 2, aspects are silent *w.r.t.* one another in AspectJ and the base program is visible to all aspects. AspectJ does not provide the notion of scope and cannot define aspects of aspects. ■

Our static analyses can take scopes into account by transforming the declarations into regular aspects. If a tagged join point is represented by a term (`tag` j `id`) then,

$$\texttt{scope}\ id\ \{id_1, id_2, id_3\}\ A$$

is transformed into A where all terms T occurring in the crosscuts of A are replaced by

$$(\texttt{tag}\ T\ \texttt{id1}) \vee (\texttt{tag}\ T\ \texttt{id2}) \vee (\texttt{tag}\ T\ \texttt{id3})$$

Analysis of strong independence as described in the previous section can be applied to such transformed aspect definitions. Independence analysis *w.r.t.* a program requires the base program and join points of inserts to be annotated similarly. Note that this encoding is for static analysis purposes only. In an actual

implementation, the join points do not need to be tagged because the identity of the current insert being executed could be recorded in the execution context.

Finally, let us mention that finer-grained scope annotations can be defined easily by allowing crosscuts and inserts to be named individually.

5 Related Work and Conclusion

Despite its importance, few work has previously been done on aspect interactions and conflict resolution.

Recent releases of AspectJ [9] provide limited support for aspect interaction analysis using IDE integration: the base program is annotated with crosscutting aspects. This graphical information can be used to detect conflicting aspects. However, the simple crosscut model of AspectJ would entail an analysis detecting numerous spurious conflicts. The reason is that the relationship between several crosscuts must be maintained by book-keeping code in advice (e.g., by incrementing a counter and check for the counter value later) [6]. In our case, this kind of relationship can (sometimes) be expressed by stateful aspects and taken into account by the analysis. In case of real conflicts, AspectJ programmers can resolve conflicts by reordering aspects using the keyword `dominate`.

DeVolder *et al.* [12] propose a meta-programming framework based on Prolog. They specify crosscuts by predicates on abstract syntax trees and define ad-hoc composition rules for specific aspects. However, this approach does not provide a general solution to aspect interaction analysis and resolution. DeVolder's work is extended by Gybels [7] to crosscut definitions depending on dynamic values (e.g. the value of a method call argument) and optimization opportunities are discussed. However, in this case the weaving process cannot be static anymore (i.e., the weaving cannot be performed by means of inserts inlining).

Andrews [1] models AOP by means of algebraic processes. He focuses on equivalence of processes and correctness of a weaving algorithm. Non-termination problems of weaving and a formal definition of `before` and `around` are discussed but aspect interaction is not treated.

Douence *et al.* [5,6] propose another model for AOP based on execution monitoring. In this model, the crosscut language is even more expressive, in fact Turing-complete, and independence or equivalence must be proven manually.

Other approaches to the formal definition of AOP — such as Wand's *et al.* denotational semantics for a subset of AspectJ [13] and, to a lesser extent, Lämmel's big-step semantics formalizing method-call interception [10] — could lead to alternative approaches to interaction analysis.

Finally, note that interaction properties arise in many fields of software engineering. For instance, Batori et al. [2] introduce "layers" which can be compared to aspects and study composition validation using semantic conditions. It would be interesting to study whether these techniques can be adapted to control semantic interactions between aspects.

We have proposed a general method for the static analysis of aspect interactions. The paper has presented three contributions. First, we have defined a

generic formal framework for AOP featuring expressive crosscuts. Second, we have given two general independence properties and have presented how to analyze them statically. Finally, we have proposed some useful commands for conflict resolution, which is based on and compatible with the presented static analyses.

As to the application of our framework, we started to formalize parts of AspectJ. This task should be completed. It would also be interesting to compare the framework precisely with the denotational semantics of Wand *et al.* [13] for a subset of AspectJ.

Other properties and analyses could been studied in our framework for AOP. For example, in some cases, the programmer may want to check that an aspect has terminated (i.e., keeps doing nothing) before another one starts. Several linguistic extensions of the aspect language are worth further study. For example, allowing crosscuts of the same aspect to share variables would make the aspect language more expressive. Also, the possibility of associating an aspect with a class or an instance would facilitate the instantiation of the framework to object-oriented languages.

References

1. J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Reflection*, pages 187–209, 2001.
2. D. Batory and B. J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)*, pages 62–87, February 1997.
3. H. Comon. Disunification: A survey. In *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, Cambridge, MA, 1991.
4. C. A. Constantinides, A. Bader, and T. Elrad. Separation of concerns in concurrent software systems. In *International Workshop on Aspects and Dimensional Computing at ECOOP*, 2000.
5. R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns*, volume 2192 of *LNCS*. Springer Verlag, September 2001.
6. R. Douence, O. Motelet, and M. Südholt. Sophisticated crosscuts for e-commerce. ECOOP 2001 Workshop on Advanced Separation of Concerns, June 2001.
7. K. Gybels. Aspect-oriented programming using a logic meta programming language to express cross-cutting through a dynamic joinpoint structure.
8. G. Kiczales et al. Aspect-oriented programming. In *Proc. of ECOOP*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.
9. G. Kiczales et al. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.
10. R. Lämmel. A semantics for method-call interception. In *1st Int. Conf. on Aspect-Oriented Software Development (AOSD'02)*, April 2002.
11. F. Nielson and H. R. Nielson. *Semantics with Applications - A Formal Introduction*. John Wiley and Sons, New York, NY, 1992.
12. K. De Volder. Aspect-oriented logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *LNCS*, pages 250–272. Springer Verlag, 1999.
13. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *FOOL 9*, pages 67–88, January 2002.